# 13 AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems

Florian Hölzl and Martin Feilkas

Institut für Informatik
Technische Universität München
D-85748 Garching, Germany
{hoelzlf,feilkas}@in.tum.de

**Abstract.** We give an introduction of the AutoFocus 3 tool[1], which allows component-based modeling of reactive, distributed systems and provides validation and verification mechanisms for these models. Furthermore, AutoFocus 3 includes descriptions of specific technical platforms and deployments. The modeling language is based on precise semantics including the notion of time and allows for a refinement-based methodology for the development of reactive systems, typically found in user-accessible embedded realtime-systems.

## 13.1 Introduction

Focus is a general theory providing a model of computation based on the notion of streams and stream processing functions [1]. It is suitable to describe models for distributed, reactive systems. Based on this mathematical semantic foundation, we have developed a CASE tool, named AutoFocus 3, to allow for graphical description of systems according to this model of computation. While Focus allows different techniques to build formal specifications of component-based, distributed systems, AutoFocus 3 only uses some of these techniques as shown in the following. Furthermore, Focus allows to use different models of time expressed through the notion of streams: in particular untimed, timed and time-synchronous streams. AutoFocus 3 is based on the time-synchronous notion of streams, which corresponds to a discrete notion of time based on globally synchronized clocks. Focus targets at precise description of applications on a logical level. Time is divided into logical ticks and logical components interact synchronously with each other in this setting.

In order to develop real distributed systems, the application must be executed on real hardware. Thus, during the development of the system, it is convenient to have several levels of abstraction and different development views. While early

[1] http://af3.in.tum.de/ provides a set of tutorials and screen shots of the current released version.

H. Giese et al. (Eds.): MBEERTS, LNCS 6100, pp. 317–322, 2010.

requirements are captured in natural language to allow flexibility, later levels use more formal specifications and finally add technical details. We believe that such a multi-layered set of models is the only way to cope with the complexity of today's systems by applying a strict separation of concerns during the phases of the development process.

AutoFocus 3, as presented here, currently covers the lower-most levels of abstraction, namely the logical system architecture and the technical architecture, which provides the application execution environment. Higher levels of abstraction and requirements oriented models have been studied in earlier versions [2]. Our goal is to provide a prototypical tool implementation that clearly distinguishes between the models of different levels of abstraction and provides support for methods to combine these models into a description of the system under development along the complete development process. Larger case studies showing the complete methodology from requirements to deployment have been published: [3] presents a case study from the field of business systems, while [4] presents a case study from the automotive domain.

## 13.2 Capabilities of AutoFocus 3

This section briefly describes the modeling techniques of the two levels of abstraction currently supported by the AutoFocus 3 tool: the logical architecture and its mapping onto a hardware/software execution platform. The logical architecture describes application specific components, while the topology describes the execution environment. For embedded systems the latter is usually a set of distributed control units and communication busses. Finally, the deployment model describes the mapping from application components onto execution and communication units.

### 13.2.1 Logical Architecture

The logical architecture defines a model of the system under development from an abstract point of view. The system's functionality is described independently of the concrete hardware/software environment and also independently of the concrete distribution of (parts of) the system on these resources.

The system consists of a set of communicating components, each having its own behavior specification, which may be stateful or stateless. Components exchange pieces of data in the form of typed messages. The semantic foundation assumes a global, discrete notion of time, e.g. the components are synchronized to a global clock.

**Component Architecture**
In AutoFocus 3 the system's model is described as a set of communicating components. Each component has a defined interface (e.g its black-box view) and an implementation (e.g. its white-box behavior). The interface consists of a set of communication ports. A port is either an input port or an output port. It is

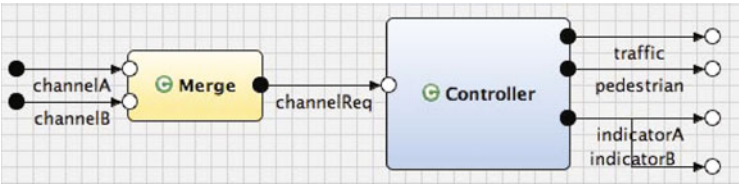


**Fig. 13.1.** A simple pedestrian crossing traffic light system

identified by its name and it has a defined type, thus describing which messages can be sent or received via this port.

Components can exchange data by sending messages through output ports and receiving messages via input ports. Communication paths are described by channels. A channel connects an output port to some input port, thus describing the sender/receiver relation. The data type of both ports must be compatible, of course. Under certain conditions, a component is allowed to send messages to itself, i.e. the model contains a feedback channel or more general a feedback loop. Output ports allow multi-cast messages, while input ports only allow a single incoming channel. From the logical point of view, channels transmit messages instantaneously.

Fig. 13.1 shows an example of a pedestrian traffic lights system, which consist of a controller for the application behavior and a merge component that merges button signal from both sides of the road. Note that AutoFocus 3 provides a hierarchical structuring of components in order to deal with larger systems in an easily comprehensible manner.

**Causality and Time**

AutoFocus 3 component networks are executed synchronously based on a discrete notion of time and a global clock. In this setting a component belongs to one of two classes. A *strong* causal component (the blue 'Controller'-component in Fig. 13.1) has a reaction delay of at least one logical time tick which means that the current output cannot be influenced by the current input values. A *weak* causal component (marked yellow in AutoFocus 3, c.f. 'Merge' in Fig. 13.1) may produce an output, which depends on the current input, e.g. the component's reaction is instantaneous. From the semantics point of view, networks consisting of strong causal components are always well-defined, e.g. for the recursive equation system induced by the channel connections unique fixed-points always exist. Component networks including weak causal components are also well-defined under the constraint that no weak-causal cycles exist, i.e. no weak causal component may send a signal that would be fed back to itself in the current time tick.

**Stateful Behavior**

To define stateful component behavior, we use a simple input / output automaton model. The automaton consists of a set of control states, a set of data state variables and a state transition function. One of the control states is defined to be

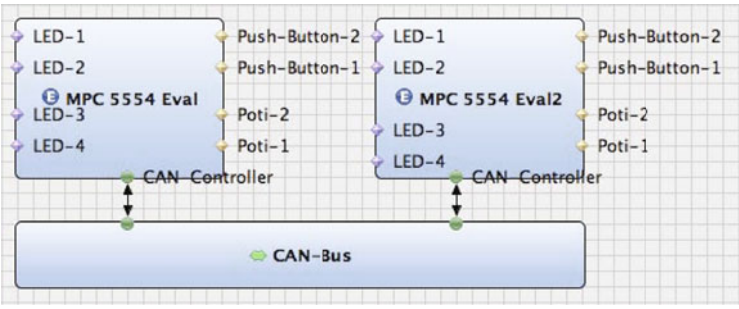


**Fig. 13.2.** Two ECU example topology for automotive lab hardware

the initial state of the component, while each data state variable has also a defined initial value. The state transition function is defined as a mapping from the current state, the current input values, and the current data state variable values to output values and subsequent data state variable values. A single transition has a source control state and a target control state, defines a set of input patterns, a set of preconditions over data state variables and variables bound in input patterns, and characterizes the output patterns and successor data state variable values.

**Stateless Behavior**
Time and again, some component has a relatively simple behavior like prioritizing certain input values or making a pre-computation. These components might not need data and control states at all. For this reason AutoFocus 3 provides a simple tabular behavior specification that gives a (possibly non-deterministic) mapping from input patterns to output patterns.

**Validation and verification**
AutoFocus 3 supports techniques to verify the logical architecture early in the development process, such as automatic test case generation and model checking. [4] presents the application of model checking techniques to verify the logical architecture. Automatic test case generation (from a separate test model) has also been applied in this case study to ensure the functional correctness of the system.

### 13.2.2 Technical Architecture

The topology architecture describes the execution environment of the system by means of execution control units (ECU) and busses. Embedded systems can observe their environment through sensors and influence it via acutators, which are connected to some ECU (like I/O devices in classical computers) or directly on some bus.

Fig. 13.2 shows an example of a topology with two ECUs connected to a common CAN bus. Each ECU provides a set of hardware ports: some LEDs, push buttons and potentiometers. These ECUs are part of our automotive lab[2]

[2] `http://www4.in.tum.de/lehre/automotivelab/`

demonstration hardware which is actively used in academic and industrial case studies and students' education [4].

**Deployment**
Having described the logical architecture and the execution environment, these two views of the system must be related to each other. In particular, each logical component has to be mapped onto some execution resource. Furthermore, logical signals need to be mapped to hardware ports, e.g. I/O devices or bus messages.

For the given example system, we could define a distributed deployment by assigning the `Merge` component to one ECU and the `Controller` component to the other. Since the `Merge` sends a signal to the `Controller` the connecting channel `channelReq` is automatically deployed on the connecting CAN bus. For the remaining signals, in particular the input signals of `Merge` and the output signals of `Controller`, we use the push buttons and LEDs, respectively.

**Code Generation**
Having completed the deployment by assigning components to ECUs and signals to hardware ports and bus messages, we can build our system to be run on the real hardware. Practically, this means to use all of the information of the model and produce C code from it, which can be compiled and flashed onto the demonstration hardware. Most of this task can be automated by suitable code generators or at least supported by the tool. Currently, this area is the most active part of our tool development activities.

## 13.3 Conclusion

We have given a compact introduction to the core features of AutoFocus 3. We have shown two system model layers: the logical architecture describing the system application by means of communicating components and the technical architecture describing the execution environment of the application by means of electronic control units, sensors, activators, and busses. We have shown a deployment model, which describes the mapping of system components to these execution resources, thus relating the abstraction levels to each other. Finally, we have obtained a complete model to be used for automatic code generation.

Of course, the model provided here is very simplistic. In particular, concerning hardware and software resources of the execution environments, we have not treated issues like precise timing and task scheduling, bus message identification and mappings of data values to bus messages. Further work must be done in this direction, possibly also including upcoming hardware techniques like multi-core processors.

AutoFocus 3 is currently implemented on the Eclipse platform[3]. Since we also develop other CASE-oriented tools, e.g. with different semantic foundations or different target domains, like machine engineering, we have built a common infrastructure, which allows a modular architecture. The deployment extension

[3] `http://www.eclipse.org/`

presented here makes heavy use of this modularity and extensibility. In detail, the topology and deployment features can be extended in specific ways: here, we have shown our automotive lab extension, which provides a event-triggered execution environment, as an example. Other extensions might include time-triggered architectures based on, e.g., FlexRay and OSEKtime. [5] already presents the formally verified mapping between the logical architecture and a time-triggered execution platform using the AutoFocus task model, which is closely related to the model of computation of AutoFocus 3.

We believe that rigorous division of the engineering process into different levels of abstraction with suitable models and precise semantics is a fundamental step towards model-based software engineering, in particular for embedded systems. We also believe that appropriate relations between these abstraction levels is vital and must be well understood from the methodological point of view, and of course supported by suitable tools. We have presented first steps towards this vision by the example of AutoFocus 3.

## Acknowledgements

We are especially grateful to Bernhard Schätz for providing continuous discussions, in particular on the language semantics and deployment questions, to Benjamin Hummel for his great work in building large parts of the tool infrastructure, and to Wolfgang Schwitzer for his work on the deployment extension of AutoFocus 3.

## References

[1] Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement. Springer, Heidelberg (2001)
[2] Geisberger, E., Grünbauer, J., Schätz: Interdisciplinary requirements-analysis using the model-based rm tool autoraid. In: Automotive Requirements Engineering (AURE 2006) Workshop at IEEE Intl. RE Conf. (2006)
[3] Broy, M., Fox, J., Hölzl, F., Koss, D., Kuhrmann, M., Meisinger, M., Penzenstadler, B., Rittmann, S., Schätz, B., Spichkova, M., Wild, D.: Service-oriented modeling of cocome with focus and autofocus. In: The Common Component Modeling Example: Comparing Software Component Models, pp. 177–206. Springer, Heidelberg (2008)
[4] Feilkas, M., Fleischmann, A., Hölzl, F., Pfaller, C., Rittmann, S., Scheidemann, K., Spichkova, M., Trachtenherz, D.: A top-down methodology for the development of automotive software. Technical Report TUM-I0902 Technical Report, Technische Universität München (2009)
[5] Botaschanjan, J., Broy, M., Gruler, A., Harhurin, A., Knapp, S., Kof, L., Paul, W., Spichkova, M.: On the correctness of upper layers of automotive systems. Formal Aspects of Computing 20(6), 637–662 (2006)